Systems and Game Technology

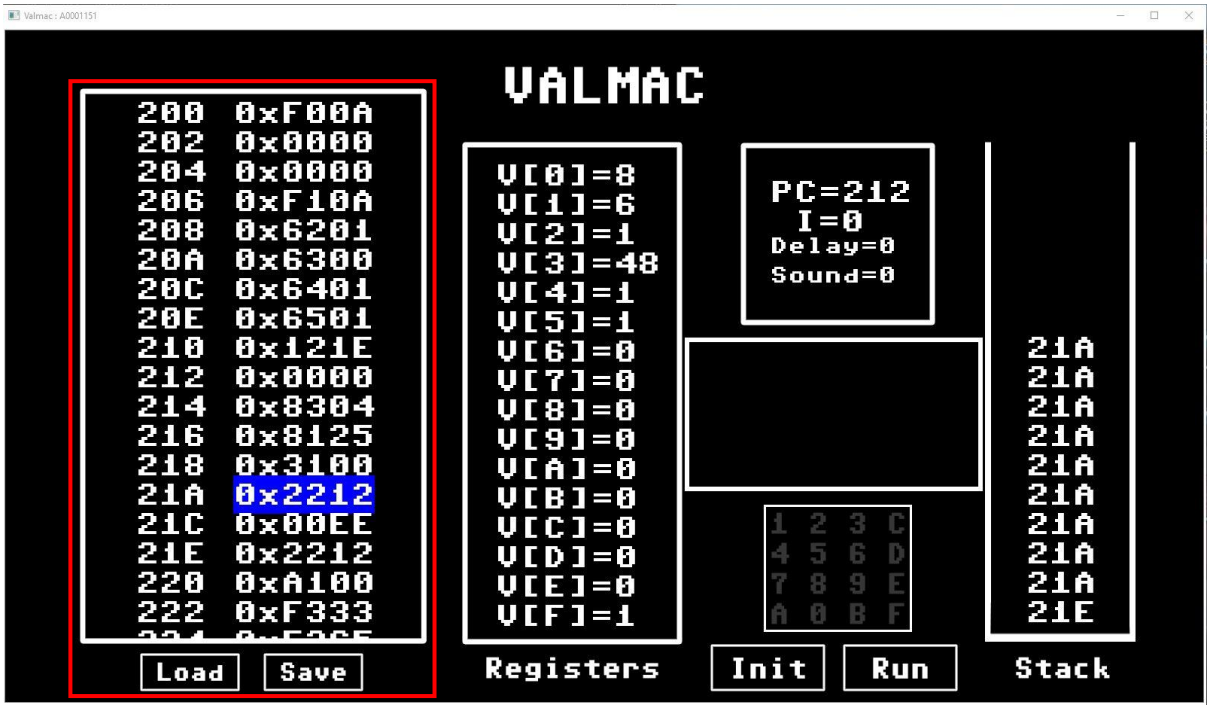
8 Bit Emulator ICA

Ivan Razvan – A0001151

1. Implementation

My project consists of an interface that allows the user to write and run programs built using the provided 8 bit opcodes. The program was designed to help the user follow the code running. As such, the interface is split in the following areas:

1.1 The Work Area:



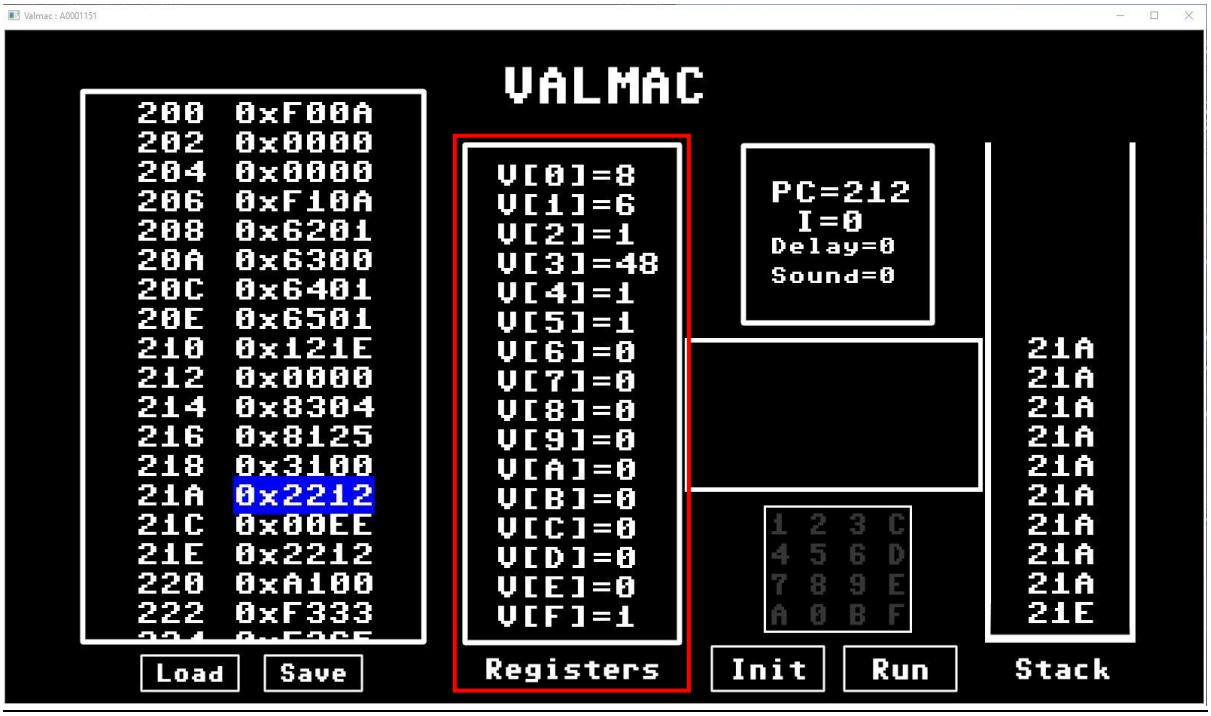
The work area consists of two columns of text, with the first showing where each line will be saved in the emulated processor's memory and the second containing the opcodes.

It will receive input from the keyboard and append only the valid characters to the code (0-9 and A-F). By using mouse clicks and scroll wheel, the user can move around the code, having the option to change it. Inserting or deleting a line is also supported through the use of Enter and Backspace keys, while a complete deletion of the code can be achieved using the Delete key.

Another implemented functionality allows the user to save and load the code to an external file called _code.txt using the designated buttons.

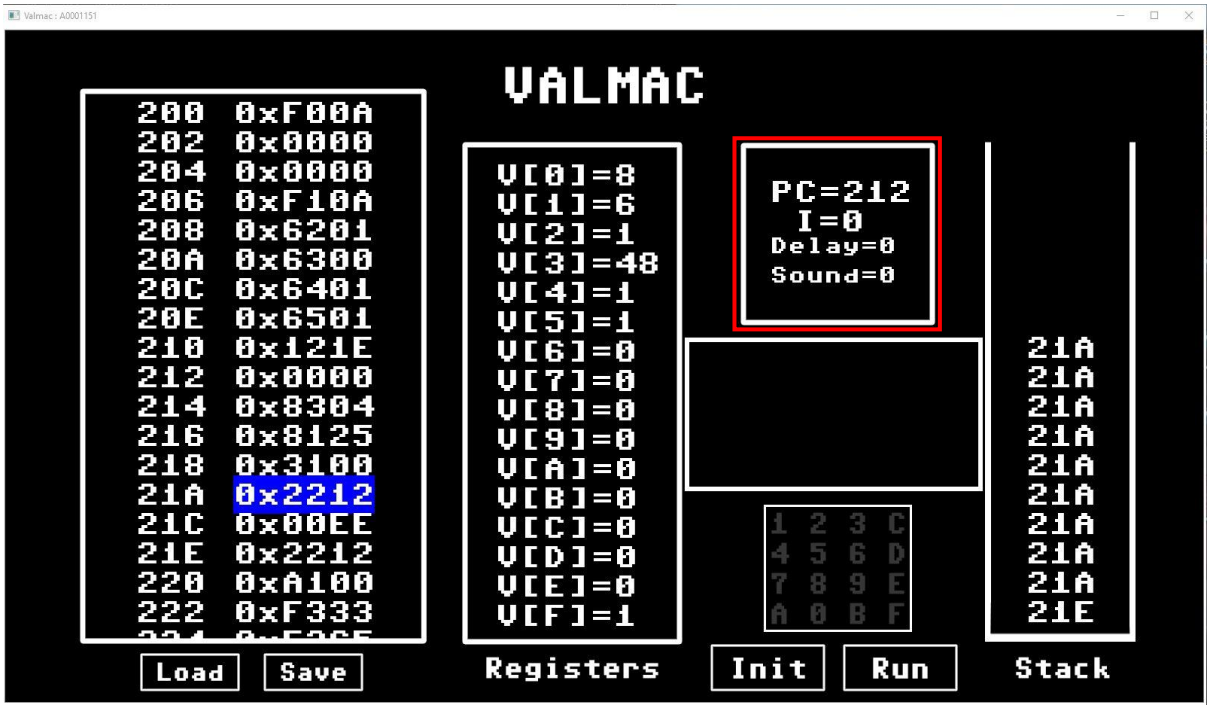
Before starting the emulator, the program checks for gaps inside the code. If all opcodes are valid, a blue rectangle will highlight the currently running opcode. During the simulation, the user is not able to alter the code.

1.2 The Registers Area:



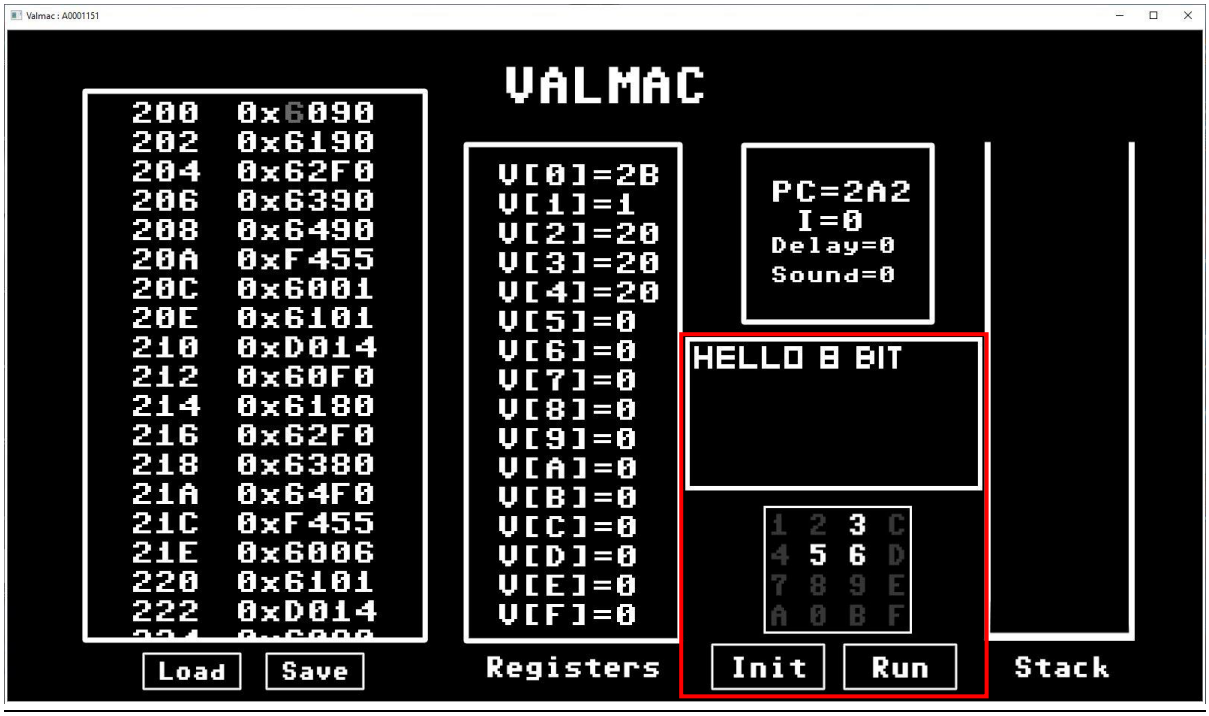
The registers area consists of a column of text, showing the value of each register.

1.3 The Miscellaneous Area:



The miscellaneous area displays the values of PC, I, delay timer and sound timer.

1.4 The Screen and Keypad Area:



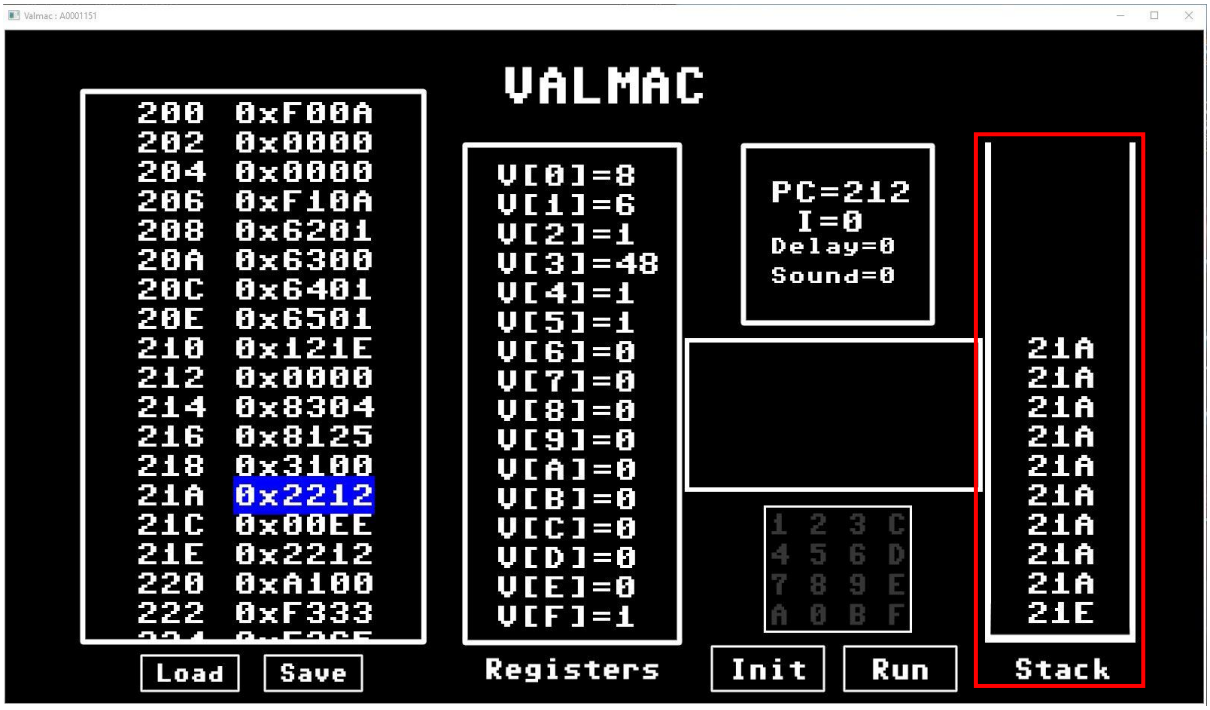
The screen and keypad area allows the user to interact with the emulator while the program is running. The screen state can be changed using the draw or clear screen opcodes, while the keypad will light up according to the keys pressed.

The run button will work only after hitting the initialise button, since the emulator needs to be reset before running any code. After that, if the code is valid, the program appends two extra exit program opcodes at the end of it. These will make sure the program stops after the last opcode read from the user (even if it is a skip next instruction).

200	0xF000	ping once
202	0x00FE	exit program
204	0x00FE	exit program

200	0x60FF	V[0] = FF
202	0x30FF	skip next if V[0] == FF
204	0x00FE	exit program
206	0x00FE	exit program

1.5 The Stack Area:



The stack area displays the state of the stack structure used to manage subroutine calls. It can render up to 16 addresses, showing where the PC will return after the end of the call. In case the user attempts to go over or below the limit, the program is stopped and one of the following messages is displayed in the console.

```
Emulator stopped: Stack underflow
Emulator stopped: Stack overflow
```

2. Low-Level Computing

Throughout the implementation of the emulator, a large portion of code consisted of low-level computing. Since the process of simulating an 8-bit processor came down to interpreting the opcodes, using bitwise operations was required.

Up until this moment, I was unaware of how much information a single number could carry. The idea of encoding data inside a number, then accessing it using masks never came up to my mind. By using the “shift” and “and” operators, I was able to extract the values stored in each nibble, then filter them using switch cases. Since each nibble carried its own piece of information, this method simplified the interpretation of every opcode, helping me write a clearer and more efficient code.

Opcode	Mask
0xXNNN	(opcode & 0xF000) >> 12
0xNXNN	(opcode & 0x0F00) >> 8
0xNNXN	(opcode & 0x00F0) >> 4
0xNNNX	(opcode & 0x000F)

3. Insights in Computer Hardware and Structure

Of all aspects of computer hardware and structure covered in this module, the one that helped me improve the most was memory management. Having to work with numbers in their binary or hex formats pushed me outside of my comfort zone, forcing me to learn how to adapt and develop upon new concepts. Understanding the concept of nibbles and endianness was crucial for me when it came to storing and reading the opcodes in the correct order.

Along with a better understanding of how numbers are stored inside memory, I also had the opportunity to manage my own stack structure, and use it to simulate subroutine calls. Up until now, I have been using functions without being aware of the actual process happening inside the computer and how my calls might end up causing the stack to overflow. But after seeing how the stack is used to control the flow of instructions, I became a bit more familiar with this type of memory management, and I can already feel the improvement in my programming skills.

4. Reflections

When it comes to developing complex applications, writing code in a high level programming language is most of the times the best option, since they allow you to easily implement complex data structures and algorithms. However, from my experience of working on this project, I realised how much low level computing will help me develop better code in the future. Although at first it was quite hard for me to find the use of this knowledge in games programming, now I have a better understanding of the tools that I will be using from now on.

Besides getting to know what is happening behind the friendly interface of the compiler, I also enjoyed developing on my own. Making sure that my program is user friendly, while also offering all the tools required to code made me think in advance about all the features I would later implement. My goal was to create an emulator that would be as accessible as possible, making it a tool for a beginner in machine code.

In conclusion, I find the experience and knowledge gained from working on this project playing a huge role in my development as a future programmer.

5. The List of Supported Opcodes

Opcode	C Pseudo
0000	nop
00E0	clear screen
00EE	return
00FE	exit program
1NNN	goto NNN
2NNN	call NNN
3XNN	skip next if V[X] == NN
4XNN	skip next if V[X] != NN
5XY0	skip next if V[X] == V[Y]
6XNN	V[X] = NN
7XNN	V[X] += NN
8XY0	V[X] = V[Y]
8XY1	V[X] = V[X] | V[Y]
8XY2	V[X] = V[X] & V[Y]
8XY3	V[X] = V[X] ^ V[Y]
8XY4	V[X] += V[Y]
8XY5	V[X] –= V[Y]
8X06	V[X] >>= 1
8XY7	V[X] = V[Y] – V[X]
8X0E	V[X] <<= 1
9XY0	skip next if V[X] != V[Y]
ANNN	I = NNN
BNNN	PC = V[0] + NNN
CXNN	V[X] = rand & NN
DXYN	draw(V[X], V[Y], N)
EX9E	skip next if key V[X] is pressed
EXA1	skip next if key V[X] is not pressed
F000	ping once
FX07	V[X] = delay timer
FX0A	V[X] = key pressed
FX15	set delay timer to V[X]
FX18	set sound timer to V[X]
FX1E	I += V[X]
FX29	I = sprite address of V[X]
FX33	store digits of $V[X]_{10}$ in memory at I
FX55	store V[0] to V[X] in memory at I
FX65	load V[0] to V[X] from memory at I

Examples of code can be found in the CodeExamples folder.

6. References

Font used: https://www.dafont.com/commodore-64.font